1. Calcolare la complessità $T(n)$ del seguente algoritmo MYSTERY1:

Algorithm 1: MYSTERY1(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
 | return 32
else
 | return MYSTERY2(n/2)+MYSTERY1(n/2)
end

function MYSTERY2(INT m) → INT
if m = 1 then
 | return 2
else
 | return 2 × MYSTERY2(m − 1)
end
```

Soluzione L'algoritmo MYSTERY1 utilizza MYSTERY2. Iniziamo quindi l'analisi da tale secondo algoritmo, che risulta essere un algoritmo ricorsivo con complessità $T'(m)$ caratterizzata dalla seguente equazione di ricorrenza:

$$T'(m) = \begin{cases} c_1 & \text{se } m = 1 \\ T'(m-1) + c_2 & \text{altrimenti} \end{cases}$$

con c_1 e c_2 costanti. Si noti che tale equazione non si basa su partizionamenti bilanciati e quindi non è possibile l'utilizzo del Master Theorem. Procediamo con il metodo dell'iterazione:

$$T'(m) = T'(m-1) + c_2 = T'(m-2) + c_2 + c_2 = \cdots = T'(1) + (m-1) \times c_2 = c_1 + (m-1) \times c_2 = \Theta(m)$$

Analizziamo ora MYSTERY1. Tutte le operazioni hanno costo costante ad esclusione della invocazione a MYSTERY2 e dell'invocazione ricorsiva a MYSTERY1. Il tempo di calcolo $T(n)$ soddisfa quindi la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + \Theta(n/2) & \text{altrimenti} \end{cases}$$

Applichiamo il Master Theorem. Abbiamo $\alpha = \frac{\log 1}{\log 2} = 0$ e $\beta = 1$. Visto che $\alpha < \beta$, per il terzo caso del teorema abbiamo $T(n) = \Theta(n^\beta) = \Theta(n)$.

2. Scrivere un algoritmo che prende in input un albero binario T e un intero positivo k e calcola il numero di nodi che si trovano esattamente a profondità k (nota: la radice si trova a profondità zero). Discutere la complessità nel caso pessimo e ottimo.

Soluzione Per risolvere l'esercizio si può usare l'algoritmo NODIK che effettua una visita in profondità partendo dalla radice e passando come parametro l'altezza k. Ad ogni chiamata ricorsiva si decrementa il valore di k fino al valore 0; in questo caso il nodo raggiunto sarà a livello k per cui dovrà essere conteggiato e la visita può essere interrotta.

Il caso ottimo si verifica se l'albero degenera in una sorta di lista (quindi se l'unico nodo ad ogni livello ha un solo figlio, o destro, o sinistro). In questo caso la complessità è $\Theta(k)$ visto che è sufficiente attraversare k nodi. Il caso pessimo si verifica se l'albero è completo fino al livello k per cui è necessario attraversare tutti i nodi fino a quel livello. Visto che un albero binario completo di livello k ha $2^{k+1} - 1$ nodi, la complessità risulta essere $\Theta(2^k)$.

Algorithm 2: NodiK(Node T, Int k) $\rightarrow$ Int

```
if T = null then
    return 0
else
    if k = 0 then
        return 1
    else
        return NodiK(T.left, k-1) + NodiK(T.right, k-1)
    end
end
```

3. Una cisterna contiene rifiuti gassosi tossici da smaltire. Lo smaltimento si effettua riempiendo particolari contenitori, ognuno avente una propria capacità possibilmente differente dalle capacità degli altri contenitori. Quando si collega la cisterna ad un contenitore per lo smaltimento del gas, il trasferimento del gas prevede di riempire completamente il contenitore, a meno che la cisterna non si svuoti completamente prima di terminare il riempimento del contenitore. Bisogna distribuire il gas nel numero massimo di contenitori possibili. Progettare un algoritmo che riceve in input la quantità K di metri cubi di gas inizialmente nella cisterna, ed un vettore V di lunghezza n tale che $V[i]$ è la capacità in metri cubi del i-esimo contenitore (i contenitori disponibili sono n). Scrivere un algoritmo che restituisce un vettore O di lunghezza n tale che $O[i]$ è la quantità di gas che verrà smaltita nel i-esimo contenitore. Si ricordi che lo smaltimento deve utilizzare più contenitori possibili. Potete assumere che la sommatoria delle capacità dei contenitori sia superiore alla quantità di gas da smaltire.

Soluzione Si può procedere secondo un criterio greedy, riempiendo i contenitori di capacità inferiore senza utilizzare (a meno che non sia necessario) quelli di capacità superiore. Questo permette di massimizzare il numero di contenitori utilizzati. Per sapere quali contenitori usare, si può inizialmente creare un min-heap contenente coppie di valori $(V[i], i)$, per $i \in \{1, \ldots, n\}$, con le capacità $V[i]$ usate come chiavi. Successivamente vengono estratti tali coppie dall'heap (quindi vengono estratte in ordine crescente di capacità) fino a raggiungere complessivamente una capacità sufficiente per contenere l'intera quantità K di gas.

L'algoritmo RiempiContenitori utilizza una struttura intermedia A che viene riempita con le coppie $(V[i], i)$ e successivamente viene costruito un minHeap H, tramite heapify che considera come chiavi i primi valori delle coppie. Successivamente si estraggono dall'heap coppie (v, j), si riempiono i contenitori indicati dall'indice j, fino a svuotamento del gas iniziale. Si usa una variabile *res* per indicare il gas rimanente dopo ogni operazione di svuotamento.

L'algoritmo esegue operazioni di costo costante ad esclusione della operazione di minHeapify e di deleteMin. La prima ha costo $O(n)$, mentre ogni deleteMin richiede tempo $O(\log n)$. Nel caso pessimo si dovranno eseguire n operazioni di deleteMin, per un costo complessivo $T(n) = O(n \log n)$.

Cont.

Algorithm 3: RiempiContenitori(Int K, Real$[1 \dots n]$ V) $\rightarrow$ Real$[1 \dots n]$

```
/* Inizializzazione: res gas residuo, O vettore in output, A coppie (V[i], i)   */
Real res ← K
Real[1...n] O
(Real,Int)[1...n] A
for i ← 1 to n do
    A[i] ← (V[i], i)
    O[i] ← 0
end

/* Generazione dell'heap contenente le coppie (V[i], i) ordinate secondo il primo campo */
minHeap[(Real,Int)]H ← A.minHeapify

/* Estrazione dall'heap dei contenitori fino a svuotamento completo del gas     */
while res > 0 do
    (Real,Int) ( v, j ) ← H.findMin()
    H.deleteMin()
    O[j] ← min(res, v)
    res ← res − v
end
return O
```

4. Progettare un algoritmo che, dato un grafo orientato pesato $G = (V, E, w)$, e due sottoinsiemi disgiunti $V_1, V_2 \subseteq V$ (disgiunti vuol dire che $V_1 \cap V_2 = \emptyset$), restituisce il cammino minimo da V_1 a V_2 ovvero restituisce il cammino di costo minimo fra tutti i cammini che partono da un qualsiasi vertice in V_1 e terminano in un qualsiasi vertice in V_2.

   Soluzione Considerando che dobbiamo confrontare tanti cammini minimi, risulta conveniente l'utilizzo dell'algoritmo di Floyd-Warshall per il calcolo di tutti i cammini minimi tra tutte le coppie di vertici di un grafo orientato pesato. Una volta calcolati tutti i cammini minimi, si controllano tutti i cammini fra coppie di nodi (v_1, v_2) con $v_1 \in V_1$ e $v_2 \in V_2$, per scegliere il più piccolo fra tutti questi. Una volta trovata la coppia (v_1, v_2) con il cammino minore, si procede a stampare il relativo cammino.

   La complessità dell'algoritmo è la medesima dell'algoritmo di Floyd-Warshall, ovvero $T(n, m) = O(n^3)$, dove n è il numero di vertici ed m il numero di archi nel grafo. Si noti infatti che le parti aggiuntive dell'algoritmo servono per ricercare i vertici $v1min$ di inizio e $v2min$ di fine del cammino minimo (con costo $O(n^2)$) e per stampare il cammino da $v1min$ a $v2min$ (con costo $O(n)$). Tali costi aggiuntivi sono inferiori in ordine di grandezza e vengono quindi assorbiti dal costo dell'algoritmo di Floyd-Warshall.

Cont.

Algorithm 4: MinimoCamminoMinimo(Grafo $G = (V, E, w)$, Set[Vertex] $V1$, Set[Vertex] $V2$,)

```
/* esecuzione dell'algoritmo di Floyd-Warshall                                  */
n ← G.numNodi()
Real D[1..n, 1..n]
Int next[1..n, 1..n]
for x ← 1 to n do
    for y ← 1 to n do
        if x = y then
            D[x, y] = 0
            next[x, y] = −1
        else if (x, y) ∈ E then
            D[x, y] = w(x, y)
            next[x, y] = −1
        else
            D[x, y] = ∞
            next[x, y] = −1
        end
    end
end
for k ← 1 to n do
    for x ← 1 to n do
        for y ← 1 to n do
            if D[x, k] + D[k, y] < D[x, y] then
                D[x, y] = D[x, k] + D[k, y]
                next[x, y] = next[x, k]
            end
        end
    end
end
/* ricerca del miglior cammino da un nodo v1 in V1 a un nodo v2 in V2           */
Int v1min, v2min, min ← ∞
for v1 ∈ V1 do
    for v2 ∈ V2 do
        if D[v1, v2] < min then
            min ← D[v1, v2]
            v1min ← v1
            v2min ← v2
        end
    end
end
/* stampa del cammino minimo da v1min a v2min                                   */
PrintPath(v1min, v2min, next)

/* procedura standard per la stampa del cammino per l'algoritmo Floyd-Warshall  */
function PrintPath(Int u, v, next[1..n, 1..n])
if ¬(u = v) ∧ (next[u, v] < 0) then
    errore(u e v non sono connessi)
else
    print u
    while ¬(u = v) do
        u ← next[u, v]
        print u
    end
end
```